#lang rina: A DSL for Describing RINA Networks

Michal Koutenský

14th February 2020

Introduction

> A programming language is low level when its programs require attention to the irrelevant.
>
> *Alan Perlis*

The phrase *programming languages* can be understood in two ways — (a) languages used to do programming; and (b) creating languages using programming methods.

This second interpretation is one of the core ideas of *language-oriented programming* [8]. In language-oriented programming, language is seen as the ultimate abstraction used to solve a problem. It encourages the development and usage of *domain-specific languages* tailored to the problem at hand. These languages are then used to solve parts of the whole problem in a clear, succinct and elegant way, and composed in a host — multi-lingual — environment.

In this article we will examine Racket [33], a programming language/ecosystem focused on language-oriented programming. We will apply the concepts of LOP in implementing a DSL in Racket for describing networks for the Recursive InterNetwork Architecture [29]. This DSL will be used to demonstrate the features Racket provides to facilitate the creation of new languages.

The article is structured as follows: section 1 on the following page provides an overview of the Recursive InterNetwork Architecture (RINA), introducing the reader to the core concepts and terminology. As the focus of this article is not on network architectures, this section is kept brief and does not concern itself with the features it provides or differences from TCP/IP besides what is necessary for the rest of the article.

Section 2 on page 3 describes the Racket programming language, its lineage, philosophy, as well as features that distinguish it from other closely-related languages. It showcases the provided language constructs as well as the overall model of the Racket ecosystem.

The final section, section 3 on page 19, walks through an implementation of a concrete DSL for RINA in Racket. The implemented DSL is evaluated in terms of features it provides and compared with existing scripts that parse this language.

The concluding part of the article summarizes the work done and the achieved results.

1 Recursive InterNetwork Architecture

> In computing, turning the obvious into the useful is a living definition of the word "frustration".
>
> *Alan Perlis*

Recursive InterNetwork Architecture (RINA) is an alternative computer network architecture described by John Day in his book *Patterns in Network Architecture: A Return to Fundamentals* [5, 29]. It aims to be a clean-slate approach to network architecture design building upon lessons learned from ARPANET, CYCLADES, ISO/OSI, TCP/IP and the problems faced when developing and maintaining these systems and architectures. In many ways, it reexamines the history of networking and the decisions made in the frame of knowledge and experience decades later, and the evolution of Internet.

The approach taken by Day is to distill and reduce networking into its purest, simplest form, as the subtitle *A Return to Fundamentals* suggests. The result of this effort can be summarized in the sentence "Networking is Inter-Process Communication (IPC), and nothing but IPC". This deceptively simple statement, when correctly understood, has profound implications on the way we think about networks and the structure of the resulting architecture.

1.1 Distributed IPC Facility

For two processes to communicate *within a system*, the system has to provide them with the necessary facilities to enable such communication. Operating systems contain several tools that allow two processes to exchange data, such as shared memory or ways to pass messages between each other.

For two processes to communicate *within a network*, the network likewise has to provide such facilities. From a process' point of view, there is (should be) fundamentally no difference between communicating within a system and across a network; all it requires is to know where to send data and where to receive data from. All the specifics of the transport method are a responsibility of the IPC facility.

When this IPC facility provides communication within a network, we call it a *Distributed IPC Facility* (DIF), and it is the core building block of RINA. DIFs are superficially similar to (TCP/IP) layers, but there are crucial differences between them. DIFs, like layers, can be stacked on top of each other; however, unlike layers, each DIF is able to provide all of the features commonly associated with the whole stack. The distinguishing feature of DIFs is not their functionality, but their scope and range. An analogy within TCP/IP would be a VPN over the Internet — it repeats (part of) the stack functionality over a different scope.

A node connects to a new DIF through another node that is reachable within a common DIF. In this way, the new DIF can be thought as sitting on top of the old one, just as the VPN sits on top of the Internet; and just like in that scenario, communication within the upper DIF happens by relaying messages through the lower DIF. An upper DIF can utilize several different lower DIFs for its communication, and various (disjoint) DIFs can be at the same level and share some or none of their underlying DIFs. The lowest DIF is the physical medium connecting two nodes, such as a wire or a wireless channel. The architecture itself

allows great freedom in combining DIFs and imposes no limits on the height of the stack.

Understanding DIFs, while necessary to understanding RINA, is not sufficient for understanding RINA. However, DIFs are the only RINA concept required to understand the RINA-related parts of this article, and as such we will end the description of RINA — and anything related to networking — here.

2 Racket

> A language that doesn't affect the way you think about programming, is not worth knowing.
>
> *Alan Perlis*

Racket [33], formerly known as PLT Scheme [34], is a programming language belonging to the Lisp family of languages. It is both a general-purpose programming language and an ecosystem for language-oriented programming [42].

2.1 The Lisp Heritage

Lisp [26] is one of the oldest programming languages still in use. Originally created in 1958 by John McCarthy [25], it has a rich history of language evolution, with many dialects exploring different approaches and ideas for programming languages. The core of Lisp are expressions and a list data structure, together with a set of seven primitive operations [26, 12]. These seven (axiomatic) operations are sufficient for describing a complete computational model, as well as implementing a Lisp interpreter.

Two of the distinguishing features of the languages belonging to the Lisp family are *homoiconicity* and a strong macro system. Homoiconicity is the quality of being able to represent the code naturally in the language's data structures.

In the case of Lisp, every expression to be evaluated is a list, potentially having another list as one of its elements in a recursive fashion. The resulting structure is in fact a tree, and can be manipulated by Lisp programs like any other data.

This metaprogramming functionality is aided by the inclusions of macros, which simplify code transformations on a syntactic level. Unlike preprocessor macros of C [18] and similar languages, which amount to little more than textual replacement of strings[1], Lisp macros have the full power of the host language at their disposal. These qualities enable Lisp programmers to create language extensions in a simple manner.

The *Revised^5 Report on the Algorithmic Language Scheme* [2], a de-facto standard describing the Scheme dialect of Lisp, specifies 23 syntactic constructs; 14 of those can be implemented as macros using more fundamental forms.

While Lisp itself does not have object-oriented features, it is possible to implement an object system using macros, and multiple implementations such as the *Common Lisp Object System* [21] exist.

Creating sub-languages using macros is common in the Lisp ecosystem. *Emacs* [10], an extensible text editor and Lisp environment written in Emacs Lisp, contains

[1] Strictly speaking, C macros operate on tokens.

a popular package `use-package` [20]. This package uses macros to create a configuration language for editor packages. Another example from the Emacs world would be `rx` [7], a macro for defining regular expressions in a more structured and readable way. *Guix* [11], a Linux distribution with declarative system configuration, likewise uses a DSL within its configuration language Guile.

2.2 Racket as a Programming Language

Racket is a general-purpose programming language. It evolved from the Scheme dialect of Lisp as an pedagogical vehicle for teaching programming in the spirit of *Structure and Interpretation of Computer Programs* [42]. It comes with its own integrated development environment *DrRacket* [6], which leverages the interactive nature of Lisp read-eval-print loops to provide a full-featured and accessible development environment.

2.2.1 Racket's Evaluation Model

Being a Lisp, Racket's syntax is based on *expressions* [43]. Expressions can be simplified to obtain other expressions. An expression that can be simplified no further is called a *value*.

```
2
```

Figure 1: A value expression.

An expression that is not a value consists of two parts, a *redex* (reducible expression) and a *continuation* (evaluation context), and may contain one or more *sub-expressions*.

```
(+ 2 0)
```

Figure 2: An expression that simplifies to a value expression 2.

In figure 2, the whole expression is a redex as it can be simplified in a single evaluation step. The continuation is empty, as evaluating the redex gives us a value.

```
(+ 5 (+ 2 0))
```

Figure 3: An expression with a sub-expression.

Figure 3, in contrast, contains both a redex and a continuation. The redex has a form of `(+ 2 0)`, while the continuation is `(+ 5 [])` where `[]` is the result of the redex evaluation. At the same time, `(+ 2 0)` is a complete sub-expression.

While every redex is a sub-expression, not all sub-expressions are redexes. As we can see in figure 4 on the next page, the expression contains two sub-expressions:

```
(+ 7 (+ 5 (+ 2 0)))
```

Figure 4: An expression with multiple sub-expressions.

`(+ 5 (+ 2 0))` and `(+ 2 0)`. Only the latter is a redex, as the former cannot be simplified in a single step.

So far, evaluating an expression required evaluating all of its sub-expressions. This is not always the case; partial evaluation is possible. Consider the conditional expression of the form in figure 5.

```
(if cond then else)
```

Figure 5: A conditional expression.

If the `cond` expression evaluates to true, the `then` expression is evaluated; otherwise the `else` expression is evaluated.

2.2.2 Language Features

When evaluating language features, it is necessary to specify that we are talking about `#lang racket` [43]. As we will see later, there are several other languages included in the core Racket environment.

We are intentionally talking about language *features*; we are not trying to classify the whole language. There are multiple commonly used categories such as *imperative*, *object-oriented*, *functional*, trying to capture the overall philosophy of the language; *with manual memory management* or *garbage-collected* describing some of the execution model properties; or even broad and vague categories such as *scripting* languages.

The problem with this kind of classification is that is unclear — and even worse — not very useful. Java [19], Python [47], and Smalltalk [37] are all object-oriented; yet one might argue that they're more different than alike. The issue becomes even more obvious when looking at some of the modern programming languages such as C# [3], Scala [44] or OCaml [27], which present themselves as *multi-paradigm*; proudly stating upfront that they cannot be easily put into a descriptive box.

Rather than this holistic stance, we will take a more reductionist approach. We shall look at languages not as a whole, but as a sum of parts; that is, *an aggregation of features* [23, 22]. Languages are a toolbox, often providing more than one way to solve a problem.

Keeping this in mind, let us try to describe Racket. Racket is a *bytecode compiled*, *garbage-collected*, *strongly typed*, *dynamically typed*, *strictly evaluated*, *lexically scoped*, *impure* language with *hygienic macros*, *modules*, *namespaces*, *higher-order*, *first-class functions*, *closures*, *tail-call optimization*, *classes*, *interfaces*, *mixins*, *traits*, *continuations*, *futures*, *places*, *green threads* and *exceptions* [43]. This feature list is not exhaustive, and while not as readable as a single-phrase label, gives us a much better overview of the tools provided to the programmer. We shall look at each of these features in more depth.

Racket is *bytecode compiled*.

Source code written in Racket is translated into an intermediate language called *bytecode*. It represents the middle ground between interpreting the source code directly and compiling to machine code, enjoying some form of benefits of both of those approaches, namely portability and performance.

Bytecode is not dependent on the execution platform; however, to run it, certain runtime environment has to be available. It is possible to bundle the runtime environment with the bytecode, effectively creating a self-contained binary to be distributed. Likewise, Racket has the option to be used like an interpreted language, directly executing the source code provided, possibly in the form of a REPL.

Racket is *garbage-collected*.

A programmer writing Racket code does not need to manage memory manually; the garbage-collection mechanism periodically looks for unused memory that can be reclaimed. Racket uses a generational garbage collector that behaves differently as the lifetime of an allocated object increases. It is possible to call the garbage collector on demand, and to specify whether all memory should be inspected or only recent allocations.

Racket is *strongly typed*.

Racket does not do any implicit type casting[2] nor has any form of type coercion; an explicit type *conversion*, denoted

```
(source-type->target-type value)
```

is required to transform the type of a value, producing a new value of the desired type. Passing a value of a wrong type to an expression results in a runtime error.

Racket is *dynamically typed*.

Passing a value of a wrong type to an expression results in a *runtime* error. Racket does not know the types of variables at compile time; types are associated with runtime values. Racket functions do not specify their return type or the types of their parameters, and it might be correct to call the same function with values of different types. Consider the following function:

```
(lambda (x)
  (cond [(string? x) (string-append x "isString")]
        [else x]))
```

This function acts as the identity function for all types except `string`. If `x` is a string, it appends "`isString`" to it. The type of parameter `x` is "anything", and in this particular case the function can correctly handle all inputs (i.e. is a complete function).

Racket is *strictly evaluated*.

Strict evaluation can roughly be described as evaluating expressions when encountered, as opposed to lazy evaluation where evaluation is delayed as

[2] Except in the case of numerical values.

long as possible (evaluated when needed). Consider an expression of the form:

```
(f (g x) (h y))
```

In a strict language, the arguments `(g x)` and `(h y)` will be first be simplified to their values, and only afterwards substituted in the body of `f`. In contrast, a lazy language will expand `f` to its body, substituting the passed arguments, and only evaluating them when their values are actually needed to progress the computation.

Racket is *lexically scoped*.
Being lexically scoped, the visibility of bindings depends on the location in the source code and the surrounding context. Several language constructs create new scope whose bindings may shadow existing bindings in the outer environment, and conversely, which hide the newly introduced bindings from this environment. An expression of the form:

```
(let ([x 1]
      [y 2])
     (+ x y z))
```

creates two new bindings for the variables `x` and `y`. These bindings are visible only within the body expression `(+ x y z)` and any previous values that might have been bound to those variables are shadowed. The body expression will use the binding for `z` from any nearest outer scope, if available.

It is worth noting that it is possible to introduce dynamic scoping in a controlled manner using the `parametrize` mechanism.

Racket is *impure*.
Although based on expression simplification, Racket is not completely pure; it has side effects. This is apparent in several language constructs.

The `begin` family of expressions allows sequencing of expressions while returning the value of the first/last one — the only way for this construct to be useful is if the other expressions have side effects.

Similarly, expressions operating on hashes have pure and impure variants; either returning a new value or mutating the passed value. While Racket does not *enforce* purity, it does *enable* leaving only small parts of the codebase impure if so desired.

Racket has *hygienic macros*.
Being a Lisp, Racket has an advanced macro system which supports hygiene. The concept of hygiene is closely related to lexical scope and binding management. To understand the purpose of hygiene, let us consider this macro:

```
(define-syntax-rule (myprint x)
  (println x))
```

As per rules for macro expansion, using our macro expands it to its body, that is

```
(let ([n 5])
     (myprint n))
```

becomes

```
(let ([n 5])
     (println n))
```

The issue with this is that the macro author intended `println` to have the meaning it had at macro definition, not at usage. We will illustrate the problem with an example.

```
(let ([n 5]
      [println
        (lambda (x) (println 'unhygienic))])
     (myprint n))
```

In its *intended use*, this code snippet should print 5. However, without hygiene, due to our redefinition of `println`, it would print the symbol `unhygienic`.

A complementary issue arises with a macro such as this:

```
(define-syntax-rule (mybegin . args)
  (let ([println
         (lambda (x) (println 'unhygienic))])
       (begin . args))
```

When called in the following fashion

```
(mybegin (+ 1 2) (println "hello"))
```

instead of printing the string `hello`, the symbol `unhygienic` will appear.

This behavior naturally follows from usage of lexical scoping and macro expansion — in the first case, the user defined binding for `println` is lexically closest. Similarly, in the second example, the macro-introduced `let` binding for `println` is the one that should be used when following the rules of lexical scope. It is clear this behavior is undesirable; it requires macro authors to know the bindings defined at macro usage and conversely, macro users to know the bindings used in macro definitions to avoid conflicts.

Hygienic macros give guarantees of safety and separation. By using the lexical scope at the time of definition (which is available to the macro expander), the behavior is made less surprising and more intuitive for users.

Racket has *modules*.

One common approach to managing complexity is decomposition. Racket structures code into *modules*. A module is a unit of code that is related and usually resides in the same file, although that is not a strict requirement.

Modules which depend on other modules for their functionality can *import* these modules. Only parts of the module that have been explicitly *exported* can be imported, providing a form of encapsulation. To import a module, the `require` expression is used, containing a relative path of the module as an argument:

```
(require "my-module.rkt")
```

An installed hierarchical grouping of modules is called a *collection*. The process of importing a module from a collection is similar to that of a normal module, but instead of a quoted relative path it uses the collection hierarchy. The following expression imports the `math` module from the `racket` collection:

```
(require racket/math)
```

Racket has *namespaces*.

The term *namespace* — as understood in languages such as C++ and C# — refers to static lexical scope. Its meaning in Racket is far more dynamic, however it is still a mapping from identifiers to bindings, as well as a module registry.

Namespaces in Racket are a first-class concept. There are functions that take namespaces as arguments and functions that return namespaces. Namespaces can be bound to identifiers and extended by being combined with other namespaces. The variable `current-namespace` always refers to a namespace that is being used to expand code, and can be overwritten to change the scope within which the code will be evaluated.

Racket has *higher-order, first-class functions*.

Higher-order functions are functions that either take functions as arguments or return a function. Racket functions are first-class values and can be manipulated like any other value. This is a powerful mechanism that allows e.g. separating structure traversal logic from the business logic that works with each element.

A traversal function `f` will take a function `g` as an argument and apply it to each element of the structure in a certain order. The author of function `f` does not need to know anything about the type being held in the structure nor possible operations on it. In fact, function `g` might not even exist at compile time and could be created at runtime by some other function.[3]

Another practical use-case for higher-order functions is as filtering predicates. A filtering function will take as an argument a function that takes an element of the collection being filtered and returns a boolean value. The sole responsibility of the filtering function is to check the predicate for each element and aggregate the results.

Racket has *closures*.

Closures refer to functions with free variables bundled with captured environment that binds those variables. They effectively allow capturing state in private variables and can be used to e.g. implement objects.

Consider the following expression, which defines a function called `count`:

```
(define count
   (let ((counter 0))
     (lambda ()
```

[3] This captures the spirit of both the *Single responsibility principle* and *Open-closed principle* found in object-oriented programming.

```
(let ((val counter))
  (set! counter (+ counter 1))
  val))))
```

This definition creates a function with a closure that captures the variable `counter`. Repeatedly calling `count` will increase the counter, even though it is not in scope any more.

```
(count)
0
(count)
1
(count)
2
```

Racket has *tail-call optimization*.

A function call is said to be a *tail-call* if it is the last action to be performed. Such a function call can be optimized to not create a new stack frame, replacing the current one instead. This is especially useful in the case of *tail-recursive* functions, which may call themselves a great number of times.

This is one possible implementation of a printing counter:

```
(define (count n)
  (letrec
      ([count-int
        (lambda (n k)
          (cond [(zero? n) k]
                [else
                 (begin
                   (count-int (- n 1) (+ k 1))
                   (println k)
                   )]
                ))])
    (count-int n 0)))
```

The call in tail position here is the call to `println` and not the recursive call to `count-int`. When calling the function with large `n`, this might result in a stack overflow.[4] Switching the two lines will allow the function to be properly optimized, bound the stack growth and handle arbitrarily large values of `n`.

Racket has *classes*.

Racket has an object system based on classes with single inheritance. Classes are first-class *values* in the language and can be bound to variables.

A class definition is an expression.

```
(class object%
  (init width height)
  (define area (* width height))
  (super-new)
```

[4] It will also count in reverse but this detail is not important.

```
  (define/public (get-area)
    area)
  (define/public (bigger-than other)
    (> area (send other get-area)))
  (define/private (grow)
    (set! area (* 2 area))))
```

This anonymous class expression showcases all core features of classes. Let us quickly go over all of them.

```
(class object%
```

The first argument of the class expression is the name of its superclass: `object%`. Similarly to other languages with classes, `object%` is the class from which all classes are derived, and can be thought of as an empty class.

The next part of the expression is the init expression:

```
(init width height)
```

It contains two arguments, `width` and `height`. These are the constructor parameters, and their values are available only during class instantiation.

What follows is a private field declaration:

```
(define area (* width height))
```

The value of area is initialized to the product of the width and height initialization parameters.

Afterwards, the superclass is initialized:

```
(super-new)
```

The `object%` class takes no initialization arguments, but if it did, they would be passed here.

The last part of the expression are method definitions:

```
  (define/public (get-area)
    area)
  (define/public (bigger-than other)
    (> area (send other get-area))))
  (define/private (grow)
    (set! area (* 2 area)))
```

The first method is a public *getter*, a method that exposes the value of a private field, possibly with some additional programming logic. The second method is a public method that compares the area of our object with another object that has a public `get-area` method. The interesting part here is the explicit message-passing: calling a method on an object is done through the `send` expression. The final expression in the class is a private method definition which doubles the object's area.

Unlike many other contemporary languages with objects, the initialization order is flexible. The sub-expressions in the class expression can be

ordered as desired to properly capture the dependencies in the initialization logic.

Instantiating a class is done through the `new` expression:

```
(new (class ...) [width 20] [height 20])
```

As you can see, we're directly passing in our class expression; our class is still anonymous. We can of course name our class to make instantiating multiple objects easier:

```
(define shape (class ...))
(define small
  (new shape [width 4] [height 3]))
(define big
  (new shape [width 13] [height 15]))
```

Besides a private—public visibility model, Racket additionally supports other method modifiers such as *abstract*, *final*, *override*, and *augment*. An *abstract* method does not include a body definition. *Final* prevents a method from being further overridden or augmented. *Override* and *augment* represent two opposing approaches to extending the functionality of a superclass method from a derived class.

Racket has *interfaces*.

Interfaces allow checking that an object or a class[5] implements a set of methods. Interfaces are extensible and transitive: if an interface *B* extends interface *A*, and an object implements *B*, it also implements *A*. Unlike classes and superclasses, an interface can have multiple superinterfaces, and a class can implement multiple interfaces.

An interface for our *shape* class that requires an implementation of the `get-area` method, called *has-area*, would look as such:

```
(define has-area (interface () get-area))
```

The first argument is an empty list, as our interface has no superinterface.

To rewrite our class to use our interface, we have to use the `class*` expression instead of `class`:

```
(class* object% (has-area) ...)
```

If our class does not implement the `get-area` method, evaluating the class expressions will result in an error.

To dynamically check whether an object or a class implements an interface, we can use the `is-a?` and `implements?` predicates.

Racket has *mixins*.

Single inheritance allows sharing an implementation only within a strict hierarchy. Mixins allow extending a class with new behavior as long as it satisfies certain conditions with regard to its exported methods.

[5] Remember, classes are values.

A `mixin` expression contains two lists of interfaces, which represent the domain and the codomain of the mixin. The mixin can be seen as a function which maps a class from the domain (implementing the given interfaces) to a new class which additionally implements the interfaces from the codomain.

To implement a *half* mixin for our `has-area` interface, we first need to define the target interface:

```
(define has-half (interface () get-half))
```

Now we define the mixin itself:

```
(define half-area-mixin
  (mixin (has-area) (has-half)
    (inherit get-area)
    (super-new)
    (define/public (get-half)
    (/ (get-area) 2))))
```

The `inherit` expression is used to get access to the `get-area` method within the expression without having to explicitly send a message.

Once we have defined out mixin, we can apply it to the `shape` class to create a `half-shape` class:

```
(define half-shape (half-area-mixin shape))
```

Racket has *traits*.

Traits are another mechanism that allows sharing implementation. While similar to mixins, traits can be combined even if they define conflicting methods. This is achieved by allowing the programmer to selectively combine trait methods and manually resolve conflicts.

The `trait-sum` expression combines two traits; `trait-exclude` removes a method from a trait; `trait-alias` copies a method under new name.

We shall define two traits: `container-trait` which has `get-volume`, `fill` and `empty` methods; and a `music-player-trait`, which has a `get-volume`, `play` and `pause` methods.

```
(define container-trait
  (trait
    (define/public (get-volume) '100ml)
    (define/public (fill) ...)
    (define/public (empty) ...)))

(define music-player-trait
  (trait
    (define/public (get-volume) '-2dB)
    (define/public (play) ...)
    (define/public (pause) ...)))
```

We would like to combine these two traits in one class that represents a container with a built-in music player. We can achieve this by aliasing the

conflicting methods and subsequently removing the original ones from the traits:

```
(define music-player+container-trait
  (trait
    (trait-sum
      (trait-exclude
        (trait-alias container-trait
                     get-volume get-container-volume)
        get-volume)
      (trait-exclude
        (trait-alias music-player-trait
                     get-volume get-music-volume)
        get-volume)
      (trait
        (inherit get-container-volume get-music-volume)
        (define/public (get-volume)
          ... (get-container-volume)
          ... (get-music-volume) ...)))))
```

Racket has *continuations*.

Continuations are a first-class value in Racket. They allow saving and replaying the execution context of a program.

Consider the expression:

```
(+ 1 (+ 1 [+ 1 1]))
```

which evaluates to 4.

The innermost expression `[+ 1 1]` is the computation, while the rest, `(+ 1 (+ 1 []))`, is the execution context — the continuation.

We shall introduce the `call/cc` expression, which passes the current continuation to its subexpression.

```
(+ 1 (+ 1 (call/cc (lambda (k) [+ 1 1]))))
```

Our lambda expression receives the continuation `k`, but as we have not used it anywhere, the result of the evaluation is still the same: 4. We can call the continuation explicitly and get the same result:

```
(+ 1 (+ 1 (call/cc (lambda (k) (k [+ 1 1])))))
```

By substituting the continuation for `k`, we get our original expression back:

```
(+ 1 (+ 1 [+ 1 1]))
```

However, we are able to call the continuation as we desire:

```
(+ 1 (+ 1 (call/cc (lambda (k) [+ 1 (k 1)]))))
```

In this case, our expression simplifies to

```
(+1 (+ 1 1))
```

skipping over one of the additions and evaluating to 3. With regards to program control flow, this is equivalent to throwing an exception and catching it in the continuation:

```
(+ 1 (+ 1 (catch [+ 1 (throw 1)])))
```

Racket has *delimited* continuations. This means it is possible to limit the size of continuations. The `prompt` expression introduces such a delimiter; the `control` expression binds the nearest continuation to its first parameter:

```
(+ 1 (prompt (+ 1 (control [+ 1 1]))))
```

One major difference between `prompt`/`control` and `call/cc` is that the `control` expression requires the continuation to be called explicitly. This can be seen in its expanded form:

```
(prompt cexpr (control k expr)) =>
(prompt ((lambda (k) expr)
         (lambda (v) (cexpr v))))
```

As we haven't called our continuation, our expression will evaluate to 3. The outermost addition will be evaluated, as will the innermost; the addition between `prompt` and `control` — the continuation — will be skipped.

Racket supports multiple continuation operators such as `call/cc`, `prompt/control` [9], `reset/shift` [4], `fcontrol` [40], `spawn` [14], `splitter` [30], `set/cupto` [13] and many others; some of these support *tagging*, to be able to refer to a specific continuation.

Racket has *futures*.

Futures are one of the constructs which provide parallel computation in Racket.

An expression of the form:

```
(future ...)
```

runs its computation in parallel to the rest of the program. However, the computation will run in parallel only if it is "future safe". This notion of safety is tied to the implementation.

Accessing a value computed by a future is done through the `touch` expression:

```
(define f (future ...))
(touch f)
```

Touching a future forces its evaluation, and repeated touches return the computed value.

Racket has *places*.

The other parallelism construct available in Racket is called places. A place is a separate Racket instance that runs it computation in parallel; however, places cannot communicate between each other except by passing messages.

This simple code snippet passes a numerical value to a place, increments it, and returns it through the channel:

```
(define channel
  (place channel
    (place-channel-put channel
      (+ 1 (place-channel-get channel)))))
(place-channel-put channel 4)
(place-channel-get channel)
```

The computation within the enclosing `(place ...)` expression will run in parallel to the main body.

Racket has *green threads*.

User-space threads, also called green threads, are threads which are not managed by the operating system; instead, they are scheduled by the program's runtime environment.

A `thread` expression creates a new thread that can execute concurrently. `thread-wait` can be used to wait until the target thread finishes executing in a blocking manner:

```
(define t (thread ...))
(thread-wait t)
```

Each thread has a corresponding *mailbox* which can be used to pass messages. Using the expression `(thread-receive)` within a thread retrieves a message from the mailbox, while `(thread-send t v)` sends the value `v` to the thread `t`.

Racket has *exceptions*.

Exceptions are a method of handling runtime errors. Whenever an error is encountered, an exception is *raised*, which stops the execution of the current expression, and instead propagates up the call stack. An exception can be *caught* by one of the outer expressions, in which case execution continues from there. If an exception is not caught by any intermediate expression, the program exits, notifying the user of the failure condition carried within the expression.

An exception is raised using the `error` expression:

```
(error "exception description")
```

An exception is usually, but does not need to be, a sub-type of the base `exn` type. Different sub-types represent different types of errors, such as a syntax error or a division by zero. By using the `raise` expression, it is possible to raise an exception of any other type than `exn`.

Catching exceptions is done using the `with-handlers` expression:

```
(with-handlers ([predicate-expr handler-expr] ...)
  body ...)
```

It contains a list of predicate-handler pairs — the predicate filters for which exceptions should the related handler be called. The body section contains the sub-expressions for which exceptions should be handled.

2.3 Racket as an Ecosystem: Beyond Macros

Racket is not just a single programming language; it is an ecosystem for multi-lingual programming. We have said that Racket is dynamically typed — however, the Racket environment ships with `typed/racket` [45] which extends the base Racket language with static[6] types. Likewise, `lazy/racket` [24] provides a version of Racket that is lazily evaluated.

These variants of Racket can be mixed interchangeably within the same codebase to allow the programmer to use the language most suited to the problem at hand. Critical sub-modules can be typed to provide some form of compile-time safety checks, while the rest of the codebase can enjoy the ease of development without having to worry about type annotations of their code.

As we have shown in section 2.1 on page 3, syntactic macros have a long history in Lisp-family languages. They have long allowed programmers to mold the language to their needs. In a certain sense, Racket with its language-oriented approach takes this idea to its logical conclusion; beyond macros.

In this section we will explore the two concepts that underpin Racket's language-oriented approach: reader macros and #lang modules.

2.3.1 Reader Macros

Reader macros are macros "one step further". Instead of transforming syntactic forms into other syntactic forms, they allow to modify the behavior of the *reader*, the part of the language that reads a text input stream and transforms it into data or syntactic forms.

Reader macros are not unique to Racket — Common Lisp supports them as well [15]. Some Scheme implementations likewise have varying levels of reader macro support [46].

By working on the input stream, reader macros allow complete redefinition of the user syntax. The DSL exposed to the user is no longer constrained by having to conform to the Lisp notion of expressions, lists, or any lexical conventions.

One form of customizing the way the reader works is through a *readtable*. The reader operates as a recursive-descent parser. The readtable maps characters to handlers that parse the content that follows.

Using the readtable is not mandatory; it is possible to supply a completely custom implementation of the reader.

2.3.2 #lang modules

Operating systems are a multi-lingual environment. Programs written in many different programming languages can be composed by the user to solve tasks in various ways.

One of the methods Unix achieved this is the concept of *shebangs*. If the first line of an executable file begins with the characters `#!`, the rest of the line is taken as a path to the interpreter used to execute that file.

Racket uses a similar concept: every source code file begins with a `#lang` declaration. This declaration specifies the set of reader and expander macros used to parse and expand the input text.

[6] More precisely, `typed/racket` is gradually typed — it allows incremental annotation of Racket code with statically checked types.

Every source file needs this declaration, else it is unclear how the source should be interpreted. However, source files within the same module need not be written in the same language.

The way these modules written in different languages, using different concepts and semantics, are able to work together is quite simple — it is Racket all the way down. After all the transformations are done, the result is a Lisp expression that gets evaluated. One writes Lisp used to write Lisp to write Lisp so they don't have to write Lisp.

The benefit of implementing a DSL is a reduction in "boilerplate" code; instead, the solution is described in a language natural and close to the problem domain.

The examples shown so far have been merely variants of Racket that changed some part of how the language works. Racket comes with another language by default, called *Scribble* [38]. Scribble is a language for creating textual documents and the accompanying toolset can be used to produce HTML, PDF or LaTeX output. All the official documentation is written in Scribble, as is package documentation for packages in the official catalog. Scribble takes the Lisp adage "code is data, and data is code" and concludes that since documents are data, they are also code.[7,8]

```
#lang scribble/base

@title{#lang rina: A DSL for Describing RINA Networks}

@section{Introduction}

The phrase @italic{programming languages} can be
understood in two ways --- (a) languages used to do
programming; and (b) creating languages using
programming methods.
This second interpretation is one of the core ideas
of @italic{language-oriented programming}.
```

Figure 6: A snippet of this document rewritten in Scribble.

Figure 6 contains the beginning of this document written in Scribble. Its target Racket implementation is hidden from the user, and the syntax resembles LaTeX more than Lisp. Nonetheless, it *is* Racket, and this can be demonstrated quite easily.

Mixing prose and the result of code evaluation is an idea adopted most notably by Jupyter notebooks, but also others such as org-mode. It is trivial to do in Scribble, as figure 7 on the next page demonstrates. Within the `@()` form, all the input is treated as a Racket expression.

[7] See also *Pollen: the book is a program* [28], another Racket document language.

[8] Another, a more subtle philosophical implication of this can be seen in Prof. Shriram Krishnamurthi — one of the core developers of Racket — only publishing his books online [39]. Krishnamurthi argues that some types of books, like computer science textbooks, should be viewed and treated as software: having a life-cycle beyond being published, incorporating bugfixes and changes based on feedback, even undergoing major revisions and restructuring. Publishing a book in print greatly limits the author's ability to update and maintain their book.

```
2 + 3 equals @(number->string (+ 2 3))
```

Figure 7: Using the output of Racket code evaluation in Scribble.

In fact, even forms such as `@italic{...}` are transformed to the Racket expression `(italic ...)` which is then evaluated to produce italic text. By leveraging the power of reader macros, Scribble is able to provide a syntax tailored to writing documents — it becomes a *domain-specific* language.

As the name suggests, domain-specific languages might not need all the power of a general-purpose language; their scope is much more limited. Every language in Racket is built on top of another language, be it Racket itself or something else. The facilities for creating languages in Racket allow the language author to pick how much of the base language they want to export in addition to their new constructs, or perhaps export certain constructs under a different name.

The Racket ecosystem has spawned a variety of languages, from practical ones, such as Pollen [28] or R^{5}RS [31] and R^{6}RS [32] varieties of Scheme, languages intended as teaching tools such as PLAI [1], to hobby languages for experimenting with language design like Heresy [41].

3 #lang rina

> When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.
>
> *Alan Perlis*

In this section, we will introduce a language used for describing RINA networks. We will follow by implementing this language in Racket and finally, evaluating the implementation when compared to existing solutions.

3.1 The Demonstrator Language

Two of the major existing RINA implementations, *IRATI* [16] and *rlite* [36], contain a common tool called the *demonstrator*. This tool takes a descriptive configuration of a RINA network and prepares a set of virtual machines and accompanying management scripts to deploy this network locally. Although the actual implementations have somewhat diverged, the configuration language is mostly compatible between these two implementations.

We will use the language description provided by the IRATI project [17], making rlite-compatible adjustments in some places.

The language is line-oriented and contains a number of declarations which all begin with a keyword. Because of this, we will limit the implementation to only two such forms, the `eth` and the `dif` form, as adding more forms to the language is an almost mechanical process with very few changes relevant to the implementation itself — the general approach stays the same.

Both forms define a DIF; the difference is what kind of DIF they define: `eth` defines the lowest DIF, an Ethernet connection between two nodes; `dif` defines a general DIF that lies on top of an `eth` or any other `dif`.

Another distinguishing feature — although an implementation detail — is that `eth`s are used to create the network interfaces for the virtual machines.

```
eth DIF_NAME LINK_SPEED NODE_NAME ...
```

Figure 8: The syntax of an `eth` declaration.

Figure 8 showcases the `eth` declaration syntax. The `eth` keyword is followed by the name[9] of the DIF, the speed of the link and the names of the nodes connected.

```
dif DIF_NAME NODE_NAME LOWER_DIF_NAME ...
```

Figure 9: The syntax of a `dif` declaration.

The syntax of the `dif` declaration, shown in figure 9, likewise begins with the name of the DIF. What follows is a name of the node which is connected to this DIF, and a list of lower DIFs which facilitate this connection.

The reason these two constructs have been chosen is that they allow us to describe an arbitrarily large and complex RINA network; their usage is meaningful even if they stand apart from the rest of the language.

3.2 Implementation

We shall begin our implementation by stating our goals, ordered by priority:

- The language should correctly parse the `dif` and `eth` forms.
- The language should validate the user input.
- The language should provide helpful feedback in case of errors.
- The language should be able to work with input where the order of dependent items is reversed.

Figure 10 on the following page shows our macro that handles `eth` forms. Our macro has a syntax parameter which we will destructure further. First, we define the expected syntax arguments: `vlan-id`, `link-speed` and `node-names`. It is necessary to distinguish the two phases during which our macro will operate — compilation time and runtime. The macro is able to evaluate expressions during compilation to produce expressions that will get evaluated during runtime.

The set of `#:attr` declarations transform the provided syntax arguments into data and binds them to variables, so that we can conveniently access the values in the expressions that follow.

The next set of declarations is responsible for *compile-time* validation of arguments. It checks the type of the provided value and prevents duplicate `eth`s

[9] Called VLAN_ID in IRATI, and had to be a number, as the VLAN tag was used as a unique identifier; this restriction has been lifted in rlite.

```
(define-syntax (define-eth stx)
  (syntax-parse stx
    [(_ vlan-id link-speed node-names)
     #:attr _vlan-id
       (eval (syntax->datum #'vlan-id))
     #:attr _link-speed
       (eval (syntax->datum #'link-speed))
     #:attr _node-names
       (eval (syntax->datum #'node-names))
     #:fail-unless
       (symbol? (attribute _vlan-id))
       "vlan-id needs to be a symbol"
     #:fail-unless
       (exact-positive-integer? (attribute _link-speed))
       "link-speed needs to be a positive integer"
     #:fail-unless
       (andmap symbol? (attribute _node-names))
       "every node-name needs to be a symbol"
     #:fail-when
       (hash-has-key? eths (attribute _vlan-id))
       "duplicate vlan-id"
     #:do
       [(hash-set! eths
         (attribute _vlan-id)
           (eth (attribute _link-speed)
             (attribute _node-names)))
        (map (lambda (node) (hash-set! nodes node '()))
          (attribute _node-names))]
     #'(begin0
       (hash-set! eths vlan-id
         (eth link-speed node-names))
       (map (lambda (node) (hash-set! nodes node '()))
         node-names))]))
```

Figure 10: The implementation of `define-eth` macro.

from being defined. Any errors produced are syntax exceptions and contain information about the syntactic context where they occurred.

The `#:do` declaration evaluates expressions during compile-time. We use it to fill the hash table used for validation with the values of the `eth` entry.

The last part is the expression to which the macro is transformed; it does the same thing as its compile-time counterpart, except at runtime. This prepared data structure will be available to the demonstrator logic to use as needed.

The macro for `dif` declarations follows a similar structure, and some structure definitions and variable declarations, such as the `eths` hash map have been omitted for brevity. The whole source code listing is included at the end as an appendix.

The next step in building our language is implementing a reader that will

transform the source text into our definition macros. Once again, we skip the definitions of some helper functions; they are trivial and should be self-explanatory based on the name.

```
(define (read-rina src in)
  (define lines (split-lines (port->lines in)))
  (let ([eths (filter (filter-cmd "eth") lines)]
        [difs (filter (filter-cmd "dif") lines)])
    (datum->syntax
     #f
     `(module rina "rina.rkt"
        ,@(map (lambda (eth)
          `(define-eth
            ',(string->symbol (cadr eth))
            ,(string->number (caddr eth))
            '(,@(map string->symbol (cdddr eth)))))
          eths)
        ,@(map (lambda (dif)
          `(define-dif
            ',(string->symbol (cadr dif))
            ',(string->symbol (caddr dif))
            '(,@(map string->symbol (cdddr dif)))))
          difs)))))
```

Figure 11: The reader implementation.

The reader implementation is displayed in figure figure 11. It splits the text into lines and sorts the input definitions into two lists: one containing `eth`s and one containing `dif`s. It then maps over these lists, producing our `(define-... ...)` macro expressions and wrapping them with syntactic context.

Every module is wrapped in an implicit `(module-begin ...)` expression. The last remaining part of our language is to create this expression using the expressions produced by the reader.

Showcased in figure 12 on the following page, this final piece of the puzzle recursively expands the syntax received from the reader into our macros. We can add additional code to each expanded definition here, as well as include any other expressions in the module. By calling a well-known function at the end of the module, the user could redefine the program logic by defining the function in their module. This function would have access to all the data structures containing parsed `eth`/`dif` declarations. The implementations showed here are the minimal variants; the actual implementation contains some additional information output.

3.3 Evaluation

Our implemented language is able to parse input files containing valid `eth` and `dif` declarations correctly.

The definitions are reordered; every `eth` is produced before any of the `dif` declarations even if the source file does not satisfy this ordering condition. Cur-

```
(define-syntax (module-begin stx)
  (syntax-parse stx
    [(_ . body)
     #`(#%module-begin
        #,@(expand-body #'body)
        (displayln "<program logic here>"))]))

(define-for-syntax (expand-body stx)
  (syntax-parse stx
    #:literals (define-eth define-dif)
    [((define-eth vlan-id link-speed node-names) . rst)
     (quasisyntax
       ((define-eth vlan-id link-speed node-names)
        #,@(expand-body #'rst)))]
    [((define-dif dif-name node-name lower-dif-names) . rst)
     (quasisyntax
       ((define-dif dif-name node-name lower-dif-names)
        #,@(expand-body #'rst)))]
    [(other . rst) #`(other . #,(expand-body #'rst))]
    [() stx]))
```

Figure 12: Definition of the implicit `module-begin` form.

rently, the order within a definition group matters — if `dif` A depends on `dif` B, A must be located before B in the source file. This is a consequence of the implementation design: dependency checks are done when evaluating the macro. A solution to this would be to evaluate the macros first and then check the consistency of the results.

Input validation is done during compilation: it is assured that the input is correct before any of the actual program logic runs. Thrown exceptions are bundled with syntactic information about the error location, containing the expression and position within the source file. The current syntactic context refers to the macro forms (`define-eth ...`) and not the input text `eth ...`. It should be possible to propagate the original syntactic context to the error reporting facilities within macros, at the cost of additional complexity. As the original forms and macro forms are almost equivalent, except the surrounding parentheses and the `define` prefix, the current approach was deemed as satisfactory compromise.

The declarations are matched on a syntactic level, and the input parsing logic is completely decoupled from the program logic.

The Python implementation included in rlite [35] parses the input in-order line by line using regular expression matching. It includes only duplicity checks as part of its parsing; type errors and dependency errors are discovered later during program operation. The errors are reported in the usual Python manner, containing the program line where they occurred — they do not clearly state the error is in the input. Understanding the cause of the error requires understanding the flow of the program itself and is unsuitable to end users.

A summary of the comparison can be found in table 1 on the next page.

Feature	#lang rina	Python
Order-independent parsing	Partial (Yes)	No
Validation before processing	Yes	Minimal (No)
Clear error location	Yes	No
Separation of parsing and application logic	Yes	No

Table 1: Summary of `#lang rina` and Python implementation comparison.

Conclusion

In this article we have explored the concept of language-based programming, which is a programming approach based on creating problem-specific DSLs and combining them in a mutli-lingual environment.

One such environment is Racket, a programming language from the Lisp family. We have shown how some of the ideas behind language-based programming have been used in Lisps predating Racket, and why the Lisp family is naturally predisposed to this kind of approach — homoiconicity.

Racket is examined twice, first as a language and later as a multi-lingual ecosystem. The features of a language are a toolbox for the programmer, allowing approaching a problem in different ways. Racket, having its roots in programming language theory community, has a mature and diverse set of features that represent a solid foundation for building purposeful DSLs.

Racket as an ecosystem leverages its unique multi-lingual approach by providing several varieties of the base Racket language by default. Along with many community maintained languages, which can all be used together, it has created an environment not unlike an operating system, but still firmly being a programming language.

The theoretical parts of this article culminate in an implementation of a DSL for describing RINA networks. It consists of a handful of macros and helper functions, as most of the heavy lifting is done by Racket itself. When compared to the current Python implementation, it provides several additional features that improve both maintainability and user-experience with very little effort.

References

[1] *1 PLAI Scheme*. URL: `https://docs.racket-lang.org/plai/plai-scheme.html`.

[2] N. I. Adams et al. "Revised^5 Report on the Algorithmic Language Scheme". In: *SIGPLAN Not.* 33.9 (Sept. 1998), pp. 26—76. ISSN: 0362-1340. DOI: `10.1145/290229.290234`. URL: `https://doi.org/10.1145/290229.290234`.

[3] *C# docs - get started, tutorials, reference. | Microsoft Docs*. URL: `https://docs.microsoft.com/en-us/dotnet/csharp/index`.

[4] Olivier Danvy and Andrzej Filinski. "Abstracting Control". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: Association for Computing Machinery, 1990, pp. 151—160. ISBN: 089791368X. DOI: `10.1145/91556.91622`. URL: `https://doi.org/10.1145/91556.91622`.

[5] John D. Day. *Patterns In Network Architecture: A Return to Fundamentals*. Prentice Hall, 2010. ISBN: 0137063385.

[6] *DrRacket: The Racket Programming Environment*. URL: https://docs.racket-lang.org/drracket/index.html.

[7] *EmacsWiki: rx*. URL: https://www.emacswiki.org/emacs/rx.

[8] Matthias Felleisen et al. "A Programmable Programming Language". In: *Commun. ACM* 61.3 (Feb. 2018), pp. 62—71. ISSN: 0001-0782. DOI: 10.1145/3127323. URL: https://doi.org/10.1145/3127323.

[9] Matthias Felleisen et al. "Abstract Continuations: A Mathematical Semantics for Handling Full Jumps". In: *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*. LFP '88. Snowbird, Utah, USA: Association for Computing Machinery, 1988, pp. 52—62. ISBN: 089791273X. DOI: 10.1145/62678.62684. URL: https://doi.org/10.1145/62678.62684.

[10] *GNU Emacs - GNU Project*. URL: https://www.gnu.org/software/emacs.

[11] *GNU's advanced distro and transactional package manager — GNU Guix*. URL: https://guix.gnu.org.

[12] Paul Graham. "The Roots of Lisp". May 2001.

[13] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. "A Generalization of Exceptions and Control in ML-like Languages". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: Association for Computing Machinery, 1995, pp. 12—23. ISBN: 0897917197. DOI: 10.1145/224164.224173. URL: https://doi.org/10.1145/224164.224173.

[14] R. Hieb and R. Kent Dybvig. "Continuations and Concurrency". In: *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*. PPOPP '90. Seattle, Washington, USA: Association for Computing Machinery, 1990, pp. 128—136. ISBN: 0897913507. DOI: 10.1145/99163.99178. URL: https://doi.org/10.1145/99163.99178.

[15] Doug Hoyte. *Let Over Lambda*. Lulu.com, Apr. 2008. ISBN: 1435712757.

[16] *IRATI Investigating RINA as an Alternative to TCP/IP*. URL: http://irati.eu.

[17] *IRATI/demonstrator: VM based (local) demo and testing tool for the IRATI stack. Documentation in README.md.* URL: https://github.com/IRATI/demonstrator.

[18] ISO. *ISO/IEC 9899:2018*. 2018.

[19] *Java | Oracle*. URL: https://www.java.com/en.

[20] *jwiegley/use-package: A use-package declaration for simplifying your .emacs*. URL: https://github.com/jwiegley/use-package.

[21] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Addison-Wesley Professional, Jan. 1989. ISBN: 0201175894.

[22] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 2012.

[23] Shriram Krishnamurthi. "Teaching Programming Languages in a Post-Linnaean Age". In: *SIGPLAN Not.* 43.11 (Nov. 2008), pp. 81—83. ISSN: 0362-1340. DOI: 10.1145/1480828.1480846. URL: https://doi.org/10.1145/1480828.1480846.

[24] *Lazy Racket*. URL: https://docs.racket-lang.org/lazy/index.html.

[25] John McCarthy. "History of LISP". In: *SIGPLAN Not.* 13.8 (Aug. 1978), pp. 217—223. ISSN: 0362-1340. DOI: 10.1145/960118.808387. URL: https://doi.org/10.1145/960118.808387.

[26] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184—195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[27] *OCaml — OCaml*. URL: https://ocaml.org.

[28] *Pollen: the book is a program*. URL: https://docs.racket-lang.org/pollen.

[29] *Pouzin society | Building a better network*. URL: http://www.pouzinsociety.org.

[30] Christian Queinnec and Bernard Serpette. "A Dynamic Extent Control Operator for Partial Continuations". In: *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '91. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 174—184. ISBN: 0897914198. DOI: 10.1145/99583.99610. URL: https://doi.org/10.1145/99583.99610.

[31] *R5RS: Legacy Scheme*. URL: https://docs.racket-lang.org/r5rs.

[32] *R6RS: Scheme*. URL: https://docs.racket-lang.org/r6rs.

[33] *Racket*. URL: https://racket-lang.org.

[34] *Racket: From PLT Scheme to Racket*. URL: https://racket-lang.org/new-name.html.

[35] *rlite/demo at master · rlite/rlite*. URL: https://github.com/rlite/rlite/tree/master/demo.

[36] *rlite/rlite: A light RINA implementation. Documentation is available in README.md (see below)*. URL: https://github.com/rlite/rlite.

[37] David Robson. "Smalltalk". In: *Proceedings of the 1983 Annual Conference on Computers: Extending the Human Resource*. ACM '83. New York, NY, USA: Association for Computing Machinery, 1983, p. 133. ISBN: 0897911202. DOI: 10.1145/800173.809715. URL: https://doi.org/10.1145/800173.809715.

[38] *Scribble: The Racket Documentation Tool*. URL: https://docs.racket-lang.org/scribble/index.html.

[39] *Shriram Krishnamurthi: Books as Software*. URL: https://cs.brown.edu/~sk/Memos/Books-as-Software.

[40] Dorai Sitaram. “Handling Control”. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. PLDI '93. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1993, pp. 147—155. ISBN: 0897915984. DOI: 10.1145/155090.155104. URL: https://doi.org/10.1145/155090.155104.

[41] *The Heresy Programming Language*. URL: https://docs.racket-lang.org/heresy.

[42] *The Racket Manifesto*. URL: https://felleisen.org/matthias/manifesto.

[43] *The Racket Reference*. URL: https://docs.racket-lang.org/reference/index.html.

[44] *The Scala Programming Language*. URL: https://scala-lang.org.

[45] *The Typed Racket Reference*. URL: https://docs.racket-lang.org/ts-reference/index.html.

[46] *Unit library - The CHICKEN Scheme wiki*. URL: http://wiki.call-cc.org/man/4/Unit%5C%20library%5C#reader-extensions.

[47] *Welcome to Python.org*. URL: https://www.python.org.